A Workflow for Training Robotic End-to-End Visuomotor Policies in Simulation

An Undergraduate Honors Thesis

Submitted to the Department of Mechanical Engineering,

The Ohio State University

as Partial Fulfillment of the Requirements for Graduation with

Honors Research Distinction in Mechanical Engineering

Submitted by:	Andrew Schellenberg
Date:	15 April 2021
Graduation:	May 2021

Defense Committee:

Dr. Haijun Su, Advisor

Dr. Wei-Lun Chao

This page is intentionally left blank.

Table of Contents

Abstract

The explicit programming methods which control most industrial robotic manipulators is a great option for precisely defined environments like factories and warehouses. These spaces are intentionally designed so robots can follow commands complete a task with limited or no awareness of their environment. But the real-world does not adhere to such strict rules; it is noisy, dynamic, and interactive. For these robots to work alongside humans in the real-world a new approach that can adapt to this randomness is needed. Research has turned to machine learning, specifically neural networks (NN), for this. Instead of programming exactly what the robot should do in every possible scenario, these methods let a NN control the robot. The NN is trained to control the robot and learns a general approach that it can adapt to whatever conditions it encounters. I focus specifically on end-to-end methods which take an observation of the environment and directly map this to a decision. These NN are trained on a specific task and run continuously. By using proprioceptive information about the robot's state and depth images from a camera in front of the robot as inputs these NN learn a visuomotor policy, akin to hand-eye coordination in humans. I share a workflow for creating these NN through behavior cloning and compare the performance of different network structures and training parameters. The workflow I present includes tools for generating demonstrations of a task, training the network, and evaluating the network. This process is designed to be adapted for different robots, tasks, or training methodologies. I show how recursive neural network structures and the training on domain randomized data both improve performance of the NN. I also describe issues where the NN do not learn the intended task and identify changes that may correct the learning process.

Acknowledgments

This project would not have been possible without the help of a so many amazing people who supported me and encouraged me on this journey.

First and foremost, thank you to Dr. Haijun Su for proving me to the opportunity to do undergraduate research with the Design Innovation and Simulation Laboratory. I am incredibly grateful for your advice and support as I took on this project. Most of all, I appreciate your patience and willingness to learn with me as we explored these neural network approaches.

I would like to thank Dr. Wei-Lun Chao not only for participating in my defense committee but also for two amazing semesters in CSE 3521 and CSE 5523 which provided a solid foundation for the machine learning aspects of this research project.

I am incredibly fortunate to have had the advice of Xianpai Zeng as well. Your consistent support in the early stages of this project was invaluable.

It would be criminal to ignore the incredible support of my girlfriend, Elena Akers. Without your encouragement I am not sure if I would have completed this project. I am beyond grateful to have had you as a proofreader: ensuring that my rambling thoughts were turned into coherent sentences. Your support in this, and in everything I do means the world.

Finally, I would like to thank my parents who have always encouraged me to aim high and provided unending love and support.

From the bottom of my heart, thank you all!

1. Introduction

1.1 Motivation

Dreams of robots that coexist in the human world as companions or assistants are nearly as old as the field of robotics itself. Certainly, no science fiction story feels complete without some robot that humans casually interact with. But in reality, robots are not that ubiquitous. Hazardous, rigorous, or simply tedious jobs exist everywhere in the world, but most robots are deployed only in warehouses and factories. Or, if they do operate in the world at large, most are specifically designed for a niche task (like iRobot's Roomba) or require human remote control. To broaden the range of environments where robots can work, research has turned to machine learning (ML) to create decision-making policies for robots to follow.

This transition is challenging because most robots are programmed in an explicit manner. They loop through commands written by their engineers to do one task in a well-defined environment. Because of the high level of control, robots in warehouses and factories trust that objects are where they should be when they should be and that their paths will be obstacle-free. If humans do work alongside robots, machine guard cages or personal protective equipment like [1] place the onus of maintaining this environment on the humans. These environments are robot-centric and careful engineering ensures that they may be assumed to be episodic and deterministic.

Human-centric environments, like homes, offices, streets, or stores, are vastly different that robot-centric ones. As humans we have learned to understand these environments but to a robot, they lack precision and repeatability.

For example, a task like restocking a grocery store's shelves require the same approach regardless of what isle or what store it is performed and what is being restocked. Humans recognize this implicitly and

learn general approaches for these tasks instead of developing a new method each time. But an explicitly-programmed robot would need a new set of instructions written be a human expert for each item it interacts with and every store it works in.

Pick-and-place, object manipulation, and inventory management tasks exist in both robot- and human-centric environments. But in the latter, each time the robot starts a task in, the positions and characteristics objects and the structure of the environment may be different. Explicit programming can handle minor variations, but this becomes infeasible as the number of cases grows. And it is difficult to imagine that the rest of the world may be constrained to the same strict rules that are enforced in robot-centric environments, so a new approach is needed.

To achieve the adaptability to work in human-centric environments, research has turned to machine learning (ML) as an alternative to explicit programming. This data-driven approach focuses on training a policy which will control the robot.

1.2 Brief Introduction to Robotic Policies

A full review of the literature is presented in Chapter 3 Related Work, this section provides a short description to give more context to the Section 1.3 Thesis Objective.

Applying ML models like neural networks (NN) in robotics has enabled researchers to train policies that can learn general methods to perform in uncertain environments. Works like [2]–[6] demonstrate successful policies using only reinforcement learning. These methods enable the robot to repeatedly attempt a task in some real or simulated environment where feedback from the environment is used to improve the policy after each attempt. Others like Zhu [7] and Rajeswaran [8] use imitation learning, where several expert demonstrations of the task are used for learning, in combination with reinforcement learning to train the policy and guide it toward a preferred approach more efficiently. Through imitation learning alone, [9] demonstrates a network for a single pick-and-place task that is robust to environmental changes mid-task.

These policies in [2]–[9] are end-to-end methods where just one NN runs continuously to predict the robot's next action. These NN are trained in simulation to learn a specific task.

In the field, there is an emphasis on comparing the performance or training efficiency between different algorithms. However, many research groups rely on tailor-made simulation environments making it difficult to compare them to a common benchmark. Often groups design their robot and task in Bullet[*] [10] or MuJoCo[†] [11] which are well-supported physics engines. But they are not designed specifically for robotics and require extra work to implement forward and inverse kinematic or path planning tools.

To address the comparison issue, James et. al [12] introduced RLBench as a common platform for designing task environments. It is built around CoppeliaSim [13] which combines common robotics tools

[*] Including [2], [6]
[†] Including [3], [5], [7], [8]

with physics simulation, which addressed the challenges of using just a generic physics engine. Many robots already have CoppeliaSim models and its user interface and PyRep [14] API for Python make CoppeliaSim easy to use.

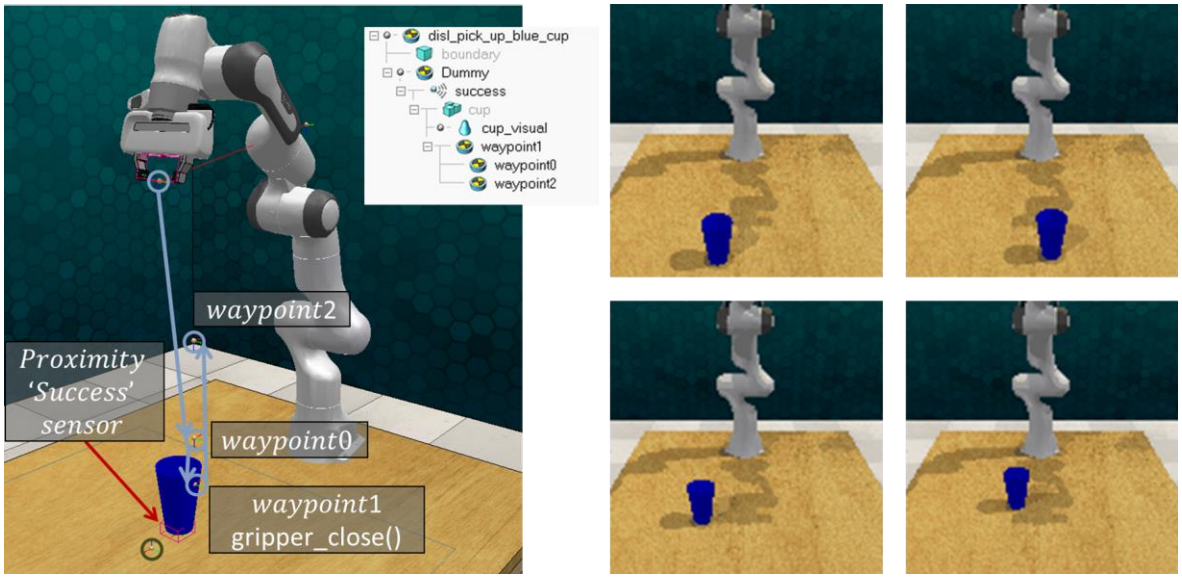


Figure 1 Task model and Initial State of Four Episodes

Figure 1 shows what a task looks like when modeled in CoppeliaSim. Because of their flexibility and documentation CoppeliaSim and RLBench were used for collecting training data and simulating the policies in this project.

1.3 Thesis Objective

The aim of this thesis is to create one contained workflow for training NN policies for robotics manipulators via behavior cloning. This includes tools for generating training data, performing imitation learning, and evaluating a policy's performance are all contained within the same code library. I achieve this by combining the existing utilities CoppeliaSim and RLBench with TensorFlow [15] for NN creation. The workflow I present is modular and can easily be expanded upon with new training algorithms, different tasks, or different robot models. This workflow is demonstrated by training a Franka Panda robot arm on two simple tasks. The networks are demonstrated and evaluated in the simulation environment to compare different network structures and training parameters. Additionally, I discuss plans for how this framework will be expanded to include reinforcement learning.

1.4 Thesis Overview

This thesis is comprised of 6 chapters. Chapter 1 Introduction introduces the research project, its motivations, and what key tools I use. Chapter 2 Neural Networks provided a brief background on the different neural network structures used in this project. Chapter 3 Related Work presents a detailed overview of the literature. Chapter 4 Methods discussed the detailed of the workflow I created and Chapter 5 Results compares the performance of the network I trained. Chapter 6 Conclusion and Future Work summarizes the findings and discussed the next steps.

This project's code is provided at: https://github.com/Schellenberg3/DISL_End_to_End_Learning

2. Neural Networks

Among the various forms of machine learning, neural networks are one of the most popular choices. These take a biologically-inspired approach to represent complex, non-linear functions.

Neural networks appear to be the most common choice for applying machine learning for end-to-end methods, and the networks can become complex. Many variations on neural networks, like convolutional neural networks, recursive neural networks, or generative-adversarial networks, are commonly put together to create endlessly complex policies.

This project too focused on training neural networks, so it is appropriate to briefly summarize some key concepts. The detail provided is intended to explain why specific structures or settings were used. References to more detailed explanations and to the mathematic basis behind these concepts are provided where appropriate.

2.1 Deep Neural Networks

Deep neural networks (DNN) are the most commonly used form of neural network (NN). These consist of several layers on neurons which pass information forward through the network. Each neuron is densely connected to next layer, meaning it passes its value to each of that layer's neurons. When passing information forward the value is scaled by a weight assigned to the connection between two neurons. This means that neuron on the next layer receives a linear combination of previous layer's values and adds a constant value called the bias; this sum is the neurons value.

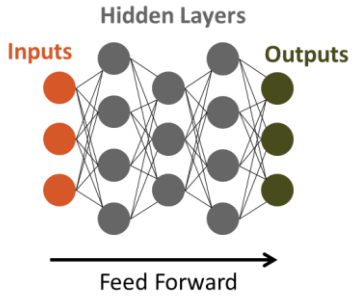


Figure 2 A Simple Deep Neural Network

This process begins with the input layer assuming the value of the data and cascades through the network to generate the predictions, shown in Figure 2 A Simple Deep Neural Network. In training, the goal is to find the proper weights for each connection and biases for each neuron.

A simple analogy for the prediction and supervised learning process of DNN is linear regression. Consider the simple case where training data in the form $D_{tr} = \{(x_n \in \mathbb{R}^2, y_n \in \mathbb{R}) \}_{n=1}^{N}$ is provided.

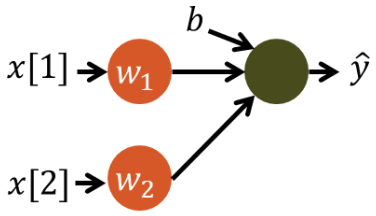


Figure 3 Linear Regression Viewed as a Neural Network

Figure 3 shows how this algorithm is visualized as a neural network. The data is linearly transformed from the input layer to the output layer where a bias is added to produce the prediction $\hat{y}$. At first, the parameters w_1, w_2, and b are randomly initialized. In the equations below, (1) describes how a prediction is made and (2) shows how the loss, measured by means squared error, is calculated.

$$\hat{y} = w_1 x[1] + w_2 x[2] + b = \theta^T \tilde{x}_n, \qquad \theta = [w_1, w_2, b], \tilde{x}_n = \begin{bmatrix} x_n \\ 1 \end{bmatrix} \tag{1}$$

$$L(w, b) = \frac{1}{N} \sum_{n=1}^{N} |\hat{y}_n - y_n|^2 = \frac{1}{N} \sum_{n=1}^{N} |\theta^T \tilde{x}_n - y_n|^2 \tag{2}$$

To learn the correct parameters gradient descent may be used. This iterative algorithm finds the parameters which minimize the error. The gradient of a single prediction is shown in Equation (3). The gradient across the entire training set is used to update the parameters θ, seen in Equation (4), scaled by the learning rate η.

$$\nabla_\theta L = \frac{2}{N}(\theta^T \tilde{x}_n - y_n)\tilde{x} \tag{3}$$

$$\theta^{(t+1)} = \theta^{(t)} - \frac{2\eta}{N}\nabla_\theta L \tag{4}$$

A similar process of gradient descent is used to find the parameters for the neurons in each layer of a deep neural network. The back propagation algorithm is used to find the gradient with respect to each layer. A detailed description of this is provided in [16].

Neural networks expand the linear regression algorithm further by applying activation functions to the output of each neuron (except for the inputs). Doing so adds nonlinearity to the network, letting it model nonlinear relationships.

Activations like sigmoid scale outputs to the range $(0,1)$ allowing neural networks to represent a probability function. Others like hyperbolic tangent scale the output to $(-1,1)$, normalizing the outputs around 0.

Another common choice is the Rectified Linear Unit (ReLU) which takes $\max(0, a)$ where a is the output. This mitigates vanishing gradients experienced by other activations when a is large and non-positive. It may also result in faster learning.

2.2 Convolutional Neural Networks

Convolutional neural networks (CNN) are a common way to apply neural networks when the inputs are images. This leverages the spatial relationship between pixels of an image. Instead of weights and biases each connecting pixel of the image to the next layer – which would grow drastically with the image's resolution – CNNs learn filters. A filter is a matrix of weights that is convoluted across the image.

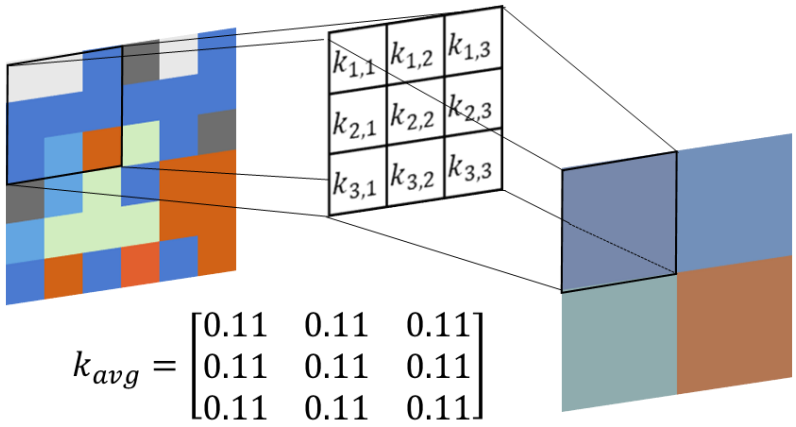


Figure 4 Averaging Filter

This convolution is shown for an averaging filter in Figure 4. The input image is treated as a tensor shaped $6 \times 6 \times 3$ and where the first dimensions describe the pixels' location and the last hold their RGB values. The averaging is applied to each of the RGB values at each location it is applied and adds the result together to get a new output pixel.

By placing the filter k_{avg} at four locations across the input image a new image of size $2 \times 2 \times 3$ is created. As expected, this image encodes the average color at each location. It also reduces the dimension of the data.

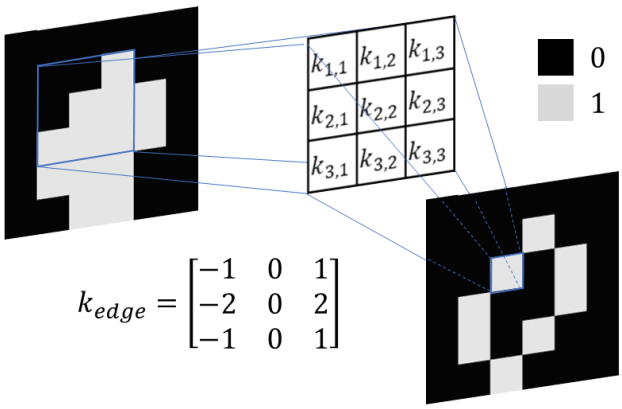


Figure 5 Edge detecting filter

Filters, like the one in Figure 5, may also be applied at each pixel on the image (with padding for those on the edge) to generate an output of the same shape. In this example the pixels only hold a 0 or 1 and the edge detection filter results in a new image that highlights the edges where pixels change.

These simple examples show the intuition behind CNN and how they can extract features like edges and reduce the dimensionality of the image. Several filters may be applied in one convolutional layer and over the course of training the CNN learns to extract specific features from data by adjusting the values in each filter.

Just like DNN, an activation function applied to the CNN's output enables it to represent nonlinear relationships. For further details for CNNs and how gradient descent can update the filters see [17].

2.3 Recursive Neural Networks

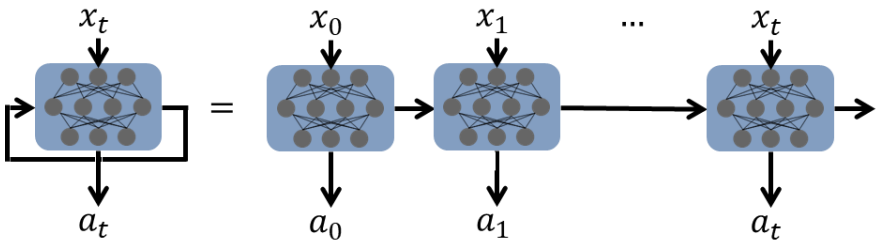


Figure 6 Recursive Neural Network

Recursive neural networks (RNN) are important for modeling time-series data like natural language processing or forecasting stock trends. This is achieved by passing the network's prediction from the previous time step forward as an input to the network at the next time step. Figure 6 shows two depictions of a recursive neural network.

By passing these predictions forward in time enables the network to remember what it has done previously. This memory leads the RNN to learn relationships between both the input x_t and prediction $\hat{y}_t$ and the order in which these appear.

In practice a specific form of RNN called long short-term memory (LSTM) is used. Instead of passing the prediction directly to the next time step, LSTM cells consist of a series of internal gates that decide what information gets passed forward. These gates are neural networks that learn what features are important and regularize it each time.

With LSTM networks it is difficult to understand what it has learned from training. This may make it challenging to understand the network's decision-making process if it performs in unexpected ways.

[18] provides an explanation of the internal logic used by LSTM networks.

3. Related Work

3.1 Machine Learning and Robotics

Most approaches for applying ML learning to robotic control problems can be categorized one of two ways: as pipeline methods, which use a series of discrete steps to generate a motion plan, or as end-to-end methods, which abstract these steps into one continuously running NN. A simplified overview of these approaches in shown in Figure 7.

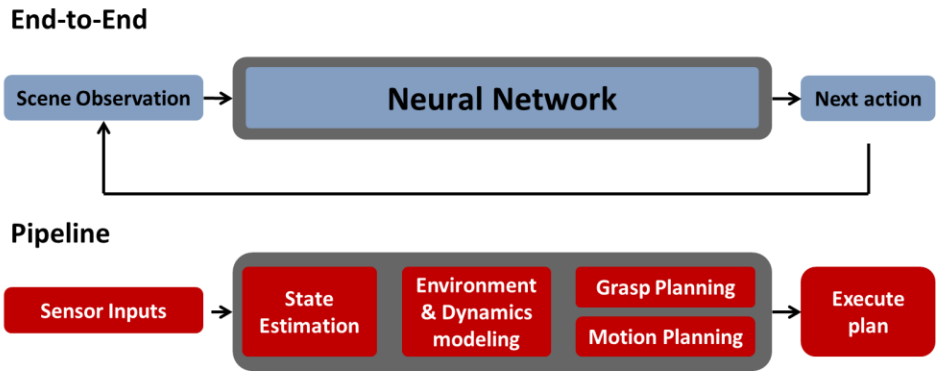


Figure 7: Comparison of end-to-end and pipeline methods

The pipeline methods do not need to consist completely of ML-based policies*, instead, these can be thought of as modules that may be swapped out with existing non-ML based methods. For example, Fang et al. [19] propose a method for selecting how a robot should grasp an object called Task-Oriented Grasping Network (TOG-Net) rates grasps both by their stability and usefulness in the context of a task. This network selects the grip and feeds this information into a motion planner or motion policy. Other groups like [20] create networks for estimating the poses – the position and orientation – of objects in the robot's environment. The intent for these pipeline methods is to create modules that replace some

* The term 'policy' rather than 'controller' denotes that the actions are decided by a probabilistic model and not a deterministic control law.

or all steps. These policies are usually shared on sites like GitHub and designed to be as task- and robot-agnostic as possible so other groups may adapt them to their needs.

End-to-end methods, the focus of this project, take a fundamentally different and biologically-inspired approach. While pipeline methods divide the process into discrete parts and produce a motion plan, these abstract the process into one policy that continuously observes the scene and decides the best action. Directly mapping observations to actions eliminates the need to make sure each module of the pipeline is properly connected and functioning as intended. This data-driven approach does introduce its own challenges: policies can be overfit, vulnerable to noise, or experience out-of-distribution inputs when tested on real-world. Training techniques must be aware of these issues and account for them.

While neural networks are not the only machine learning technique, they are a popular choice for end-to-end policies. Unless otherwise noted, from this point on the term 'policy' implies a neural network.

One network structure can be trained on many different tasks [7]. Hausman et al. [21] demonstrate a training technique that enables a policy to learn variations of a task and perform a specific variation based on an intent provided to the policy. In [9], James et al. show training techniques which help a policy trained in simulation transfer to a real-world system. They also show that the system can adapt as the location of key objects and distractors change mid-task.

Unlike pipeline methods, groups tend to share only the details of their training process and not the actual trained policy.

A useful way to view tasks is as Markov decision process (MDP), where the next state of the environment is determined only by the current state and the action taken. Under this assumption, policies are expressed as:

$$a^{(t+1)} = \pi\left(s^{(t)}; \theta\right) \tag{5}$$

Where the policy π decides the action $a^{(t+1)}$ based on $s^{(t)}$ the observed state where t is the current time step. Taking the action $a^{(t+1)}$ leads to the next state, $s^{(t+1)}$. The goal is to learn the proper parameters, θ, for the policy.

It should be noted that networks with LSTM units technically do not fit the definition of an MDP. This is because their decisions are dependent on both the current input and on the contents of the memory from previous states. Despite this, an MDP is still a convenient way to imagine the interaction between a policy and its environment.

For visuomotor policies, the input state consists of image(s) of the scene and proprioceptive information about the robot. In a sense, these policies are designed to learn what humans would consider hand eye coordination. Some groups have successfully used eye-in-hand images [2] but many use a stationary camera in front of the robot. For proprioceptive information, some groups use joint velocity or position and some information about the gripper's state. The output action is usually in the same format as the proprioceptive information – if joint velocities are the input, then they are also the output – but this is also not always the case.

The challenge with end-to-end methods is that they do not generalize well to systems and tasks different than the one they were trained on. For this reason, most groups share their training methodology instead of the networks themselves. The expectation is that the effort required to retrain a network for a new task would be equivalent to training a new network from scratch. For this reason, training networks efficiently (minimizing training time and data) is a key research area.

3.2 Training Methods

Because training is so important to the field of end-to-end robotic policies, it is worth discussing the main methods in detail. Three approaches are commonly used: imitation learning (specifically behavior cloning), reinforcement learning, or a combination of the two.

It is possible to collect training data from or perform reinforcement learning on real robot systems. For example, in [4] a door opening task is learned over the course of several hours of real-world attempts. Training in the target environment avoids the sim-to-real transfer problem, described in 3.3 Sim-to-Real Transfer.

However, for many tasks it may be difficult to accurately measure the real environment in episodes or reset it between them. Using a simulation allows the training or demonstration collection to be scaled without the need for more hardware. Simulation is also a safer environment for testing policies with little training which might move in unpredictable ways, damaging the hardware, environment, or humans.

3.2.1 Imitation Learning

This method focuses on training a policy to mimic the actions on a 'expert' demonstrator. This expert provides a data set of examples. This data set contains multiple demonstrations, also called episodes, of a task with a tuple of observation and action at each time step.

In this project, I focus on the technique of behavior cloning where the data provided by the expert demonstrations is used to train the policy to predict the same action sequences as the expert given the same initial input. However, there are other techniques for imitation learning, like trying to learn reward

function from the demonstrations. Hussein et al. [22] provide a detailed review of imitation learning methods applied to games and robotics.

For behavior cloning, the expert episodes could come from a human directly controlling the robotic manipulator or by an inverse kinematics-solver moving between pre-determined waypoints. The latter is the case for examples generated with RLBench.

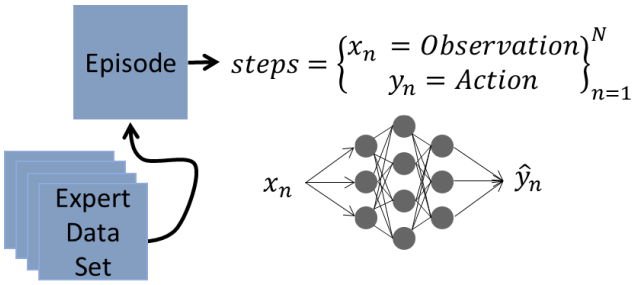


Figure 8: Behavior Cloning Imitation Learning

The general approach is summarized in Figure 8. The features x_n and labels y_n make this a supervised learning method, simplifying the learning problem. Well-known techniques like stochastic gradient descent may be applied to train the network.

Mimicking the expert data means that the network will learn to perform the task in the same manner as the expert. This allows the network to be guided towards a preferred and expected solution.

A data set of *only* expert examples means that the network never sees a failed attempt of the task. While this may seem ideal, it mean that the network may not learn how to recover if it reaches an unexpected state.

Training a policy through behavior cloning alone is possible [9], but it typically requires an *immense* number of demonstrations. In this project, RLBench allows us to solve this problem by procedurally generating these expert demonstrations.

3.2.2 Reinforcement Learning

If imitation learning is learning by observation, then reinforcement learning is learning by doing. In this method, an agent interacts with the environment based on the action selected by its policy.

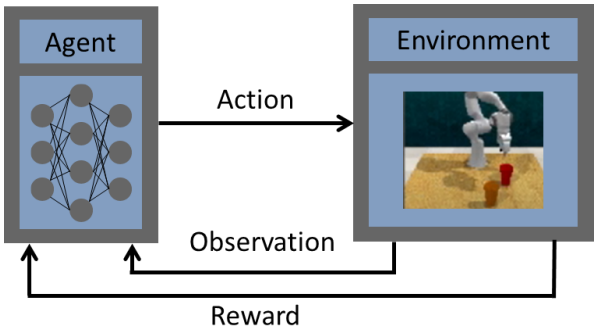


Figure 9 Reinforcement Learning

At each time step the agent decides an action to perform on the environment based on its current policy, and the environment provides an updated observation for the next decision. This repeats until the agent completes the task or reaches a maximum number of steps. The environment then provides a reward describing how well the agent did on the task. The agent uses this to update its policy and attempts a new episode of the task. This process is described in Figure 9 and repeats until the agent is unable to increase its reward.

Using reinforcement learning seems to be the most common approach for training. Many reinforcement algorithms have been successfully applied for this including Recurrent Deterministic Policy Gradient (RDPG) [3], Q-Learning with Normalized Advantage Functions (NAF) [4], Proximal Policy Optimization (PPO) [2].

Designing reward functions can be a challenge. It is easy to give a very sparse reward where a score of 1 is provided if the task was completed successfully and 0 otherwise. But this makes it challenging for the network to understand what actions at what time led to the reward. This causes the policy to spend a longer time exploring in the environment to learn the correct policy.

Conversely, complicated rewards might lead the agent to converge on a sub-optimal or incorrect policy. Consider the case where the task is to pick up a cup, and the reward function is based on the time it takes for the agent to reach the cup and how far the cup moves from its starting position. Through random exploration, the agent might learn that the easiest way to maximize its reward is simply sweeping the cup off the table. Careful consideration is required to create a good reward.

3.2.3 Combined Learning

While imitation and reinforcement learning alone can produce successful policies, using them in combination can leverage the advantages of each while addressing their inefficiencies. Providing only a small number of expert demonstrations, a manageable amount to generate by hand, may reduce the training time of reinforcement learning and increase reward that the final policy can earn.

For example, [8] pre-trains their policy for a 24 Degree-of-Freedom on 25 demonstrations before performing reinforcement learning with a Natural Policy Gradient (NPG) method. For a few of the tasks considered, policies trained with reinforcement learning alone could not learn a policy to successfully complete the task oven once.

The expert demonstrations do not have to be used for behavior cloning. In [2] they provide an adaptive curriculum for the agent. When the agent starts learning it assumes control of the demonstration at a time step near the end of the task. For example, in a block stacking task the agent might begin at the point in the demonstration where the expert had already picked up the block. As the policy improves, it starts further away from the end goal. Using the examples in this manner allowed them to achieve much higher success rates faster than other reinforcement methods.

3.3 Sim-to-Real Transfer and Robust Policies

Training in simulation has many benefits. Its scalability is dependent only on the computational power available, environments can easily be edited, and information (like the exact position and orientation of all objects) is easily available. But despite these benefits that simulation training provides, it creates an additional requirement that the policy be robust enough to perform well on the real system which will inevitably differ from the simulations. Even though the simulation are modeled after the real environment, modeling error or noise present in the real-world mean that these states experienced by the policy well learning will not be the same as the real-world environment. This means that we must also solve a transfer learning problem if we wish to deploy a simulation-trained policy on real hardware.

One approach for this is the one- or few-shot transfer. A policy is first trained to completion in simulation and then trained further on the real system. The assumption is that a policy which is successful in simulation will still work well on the real system even if it is not perfect. By starting with this simulation-trained policy, fewer training iterations may be required for the policy to learn the correct approach. This process alleviating some of the concerns of on-robot training mentioned at the end of 3.2 Training Methods.

However, many groups focus on directly transferring the policy without additional training. This is referred to as zero-shot transfer. For example in [7], a simulation-trained policy was successfully deployed on a real system with a similar success rate to its attempts in simulation. Identifying training techniques for zero-shot transfer is an active research area.

In theory, transfer could be achieved with a hyper-realistic model. But effort required to create the model and computation power required to simulate the environment mean that this has diminishing returns after a certain amount of complexity.

A more common approach for creating robust policies is domain randomization. This consists of both visual randomization and dynamics randomization which vary the appearance and dynamic properties of the environment, respectively.

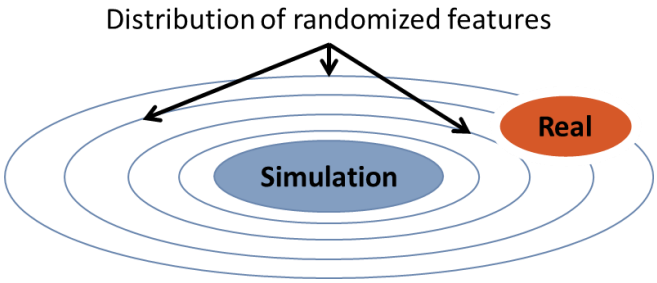


Figure 10 Feature state expansion with domain randomization

Figure 10 shows the intuition behind domain randomization. Randomizing the feature space of the simulation expands the possible features that the network will see during training.

For dynamics randomization, parameters like an object's mass and friction and the joint's properties like maximum speed and torque are varied between episodes. This is most appropriate for reinforcement learning where the network learn must learn a policy that can tolerate this variation. This mitigates modeling errors and perhaps errors from signal noise on real hardware.

Visual randomization may be used in both reinforcement and imitation learning. This technique adjusts the visual appearance of all objects that are unrelated to the task. This highlights the task-relevant objects across episodes and prevents the policy from learning spurious correlations between irrelevant features in the environment. Additionally, by learning to ignore the look of these features in simulation, like the table and wall, the policy will ignore these features in the real-world. This means that the policy could be transferred to a real system whose environment is visually distinct from the one modeled. From the policy's perspective, the real-world is just another noisy background.

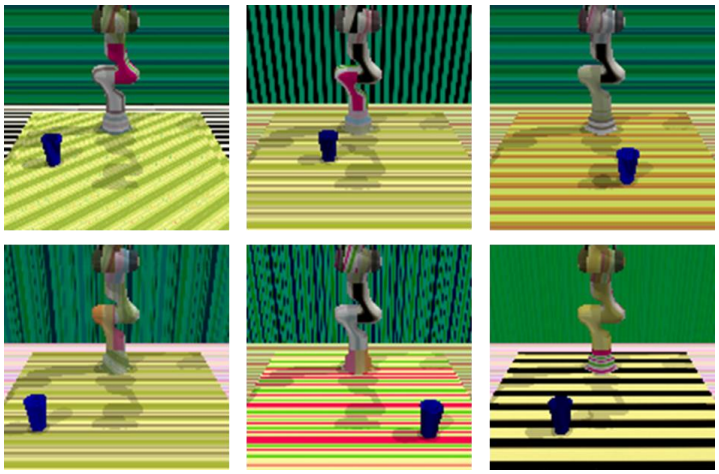


Figure 11 Visual Domain Randomization

Some examples of a visually domain-randomized task are shown Figure 11. In this task the robot needs to identify and pick up the blue cup. Note that only the cup, the focus of the task, remains constant across the episodes while all other surfaces are randomized.

When applying visual randomization, one might also adjust properties of the lighting or the position of the lights and camera. However, this was not done in this project.

Adding random distractor shapes to the environment is another way to improve the robustness of a policy. In [9], this is used in combination with all of the visual domain randomization techniques mentioned to train a policy on behavior cloning alone. Deployed on a real robot, their policy was robust and performed successfully in the real-world despite changes in lighting, camera position, and the placement of distractors while completing a task.

For reinforcement learning specifically, one strategy is to apply random forces to the joints enabling the policy to learn to reject those disturbances. [2] presents a strategy for gradually introducing domain randomization as the network's performance improves. This, along with adaptive curriculum, allows their network to achieve zero-shot transfer.

Zhao et al. [23] provide an excellent review of techniques used for sim-to-real transfer.

4. Methods

Many of the groups discussed under Chapter 3 Related Work have years of experience creating end-to-end policies and, in the process, have developed their own code and simulation libraries. However, most of this code is internal to these groups and even that which is shared is often custom built for a single robot or a few specific tasks. While these could be adapted to include desired robots or other tasks, they generally lack detailed documentation on how to do so.

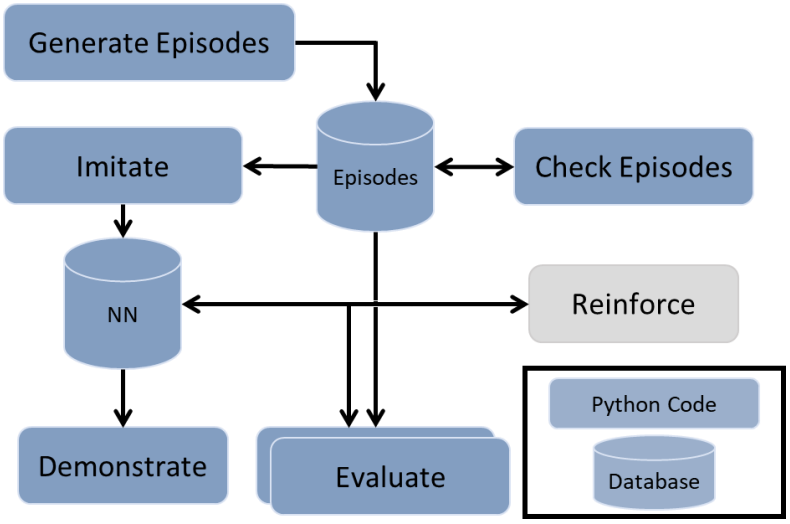


Figure 12 Workflow Overview

I addressed this need of a flexible process by writing a simple end-to-end training workflow, shown in Figure 12. This code is modular so it can be adapted to a wide range of robots, tasks, and algorithms. Each module in the process is a Python script. Databases are used to save training data or trained networks on disk so they can be utilized by different modules. The arrows show how these modules and databases can share information with each other.

The greyed out Reinforce module represent the intended placement of reinforcement learning code that will be developed in future work.

CoppeliaSim provided all physics simulation and visualization for the workflow. It comes with models for a wide range of robotic systems and is well documented. Its graphical interface makes it simple to create new models of robots, custom grippers, or other objects.

RLBench expanded on CoppeliaSim by providing a simple way to build tasks and interact with the simulation environment. It too is a critical part of the workflow. Because this tool is designed for reinforcement learning, a few functions were customized to better suit behavior cloning.

I utilized TensorFlow to create and train neural networks. It's functional API make it easy to specify multi-unput and multi-output NN.

By combining these tools, this workflow provides support for the entire model creation process: generating training data, performing supervised leaning on that data, and evaluating trained models. The modules provide a command-line interface for the user.

The following selections provide an overview of how these modules were used and the actual code for this project is available on GitHub[*].

[*] Code may be found at: https://github.com/Schellenberg3/DISL_End_to_End_Learning

4.1 Network Structure

In this project I explored the performance of several variations on the same base network structure. The evaluation of these networks and comparison of training processes are discussed in Chapter 5.2 Effects of Network Structure. Here, a detailed description of the networks is provided.

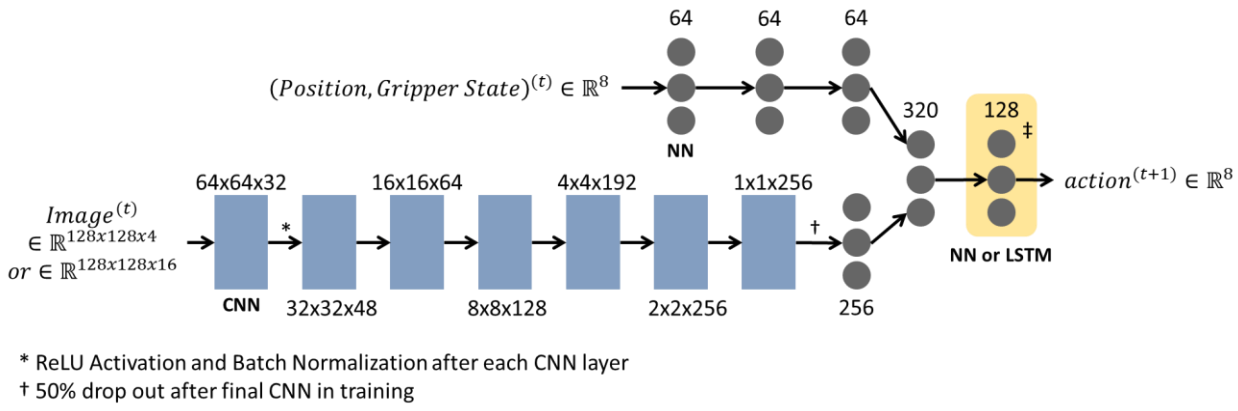


Figure 13 Base Neural Network

The network shown in Figure 13 consists of two branches: one for proprioceptive information which goes through several dense layers and another for the visual input which passes through a series of CNN. The outputs of these networks are concatenated together and passed through a third layer which is either another dense layer or a LSTM.

After each convolutional layer, a ReLU activation is performed followed by batch normalization. The ReLU adds nonlinearity to the model. Batch normalization re-centers and re-scales the data and provides stability during training.

In training a drop out is applied after the final CNN layer[*]. This sets a random 50% of the neurons to zero. This practice is commonly used to ensure that the network learns to encode a relevant feature for

[*] Future work may not include this. Drop out layers may have a negligible or detrimental effect on regression.

of the neurons. Because it does not know which neurons will be set to zero, it cannot become overly dependent on just one feature.

I am interested in two variations of this structure. I investigate using just the current image as the visual input versus using a stack of the four most recent images, a technique proposed by [9]. The effect of the LSTM layer is also explored by swapping this layer with an equivalent-sized dense layer.

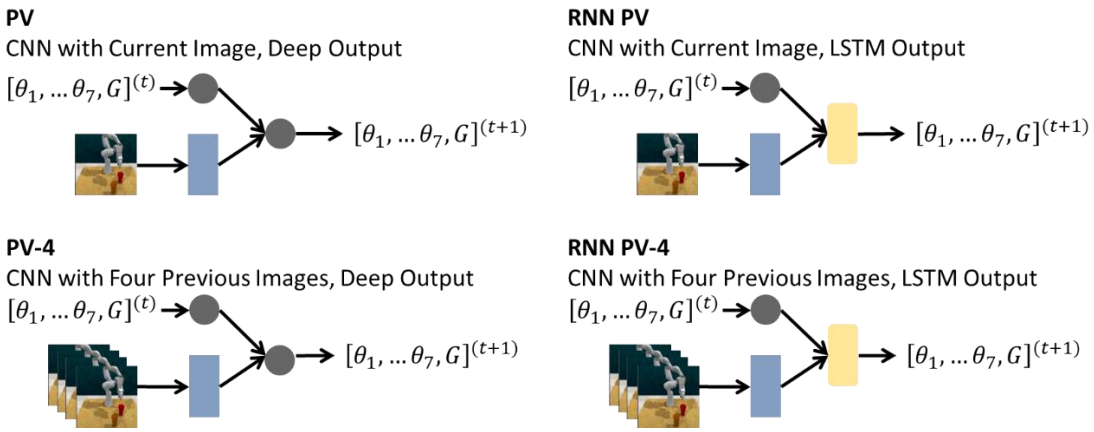


Figure 14 Variations of the Base Neural Network

The network variations and the shorthand for them are seen in Figure 14. A network similar to RNN PV-4 is successfully trained by behavior cloning in [9] and a network like the PV one is trained by reinforcement learning with imitation learning in [2]. Through these networks, I seek to understand the importance of the LSTM layer and how much additional insight a series of images provides.

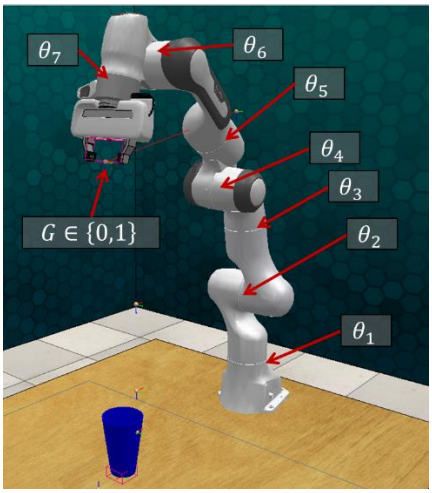


Figure 15 Proprioceptive Inputs

The joints of the robot act as an input to one branch of the neural network. Figure 15 Proprioceptive Inputs shows how these joints are labeled on the robot. For the purposed of formatting the data, each joint is considered to have a range of motion of $[-\pi, \pi]$.

All networks use a mean-squared-error loss. Each term is weighed the same and the gripper state is treated as a continuous value. Equation (6) show the equation for this loss:

$$L = \frac{1}{8}\left[\sum_{i=1}^{7}\left(\theta_i - \hat{\theta}_i\right)^2 + \left(G - \hat{G}\right)^2\right] \tag{6}$$

This MSE loss was chosen for simplicity. However, future work may use a cosine similarity loss function could be better for the joint values. Additionally, the gripper prediction would be best expressed as a categorical variable describing what action to take: open, close, or no action. As discussed later in Section 5.4 Demonstrations, a more complex error term that incudes other predictions about the robot's state or environment may be desirable for future work.

4.2 Dataset Creation

RLBench is published with over 100 tasks already defined. These range from simple tasks like reaching a floating target in front of the robot to complex ones like closing a set of windows. I considered two different tasks in this project, the aforementioned reach target task and a custom cup pick up task.

Creating a data sets begins by creating a CoppeliaSim model of the task and an associated Python file in RLBench. One reason RLBench was selected for this project was the amount of documentation that it provides for this process*. The task's model defines how the expert demonstrations are collected so it is worth discussing how the cup pick-up task is structured.

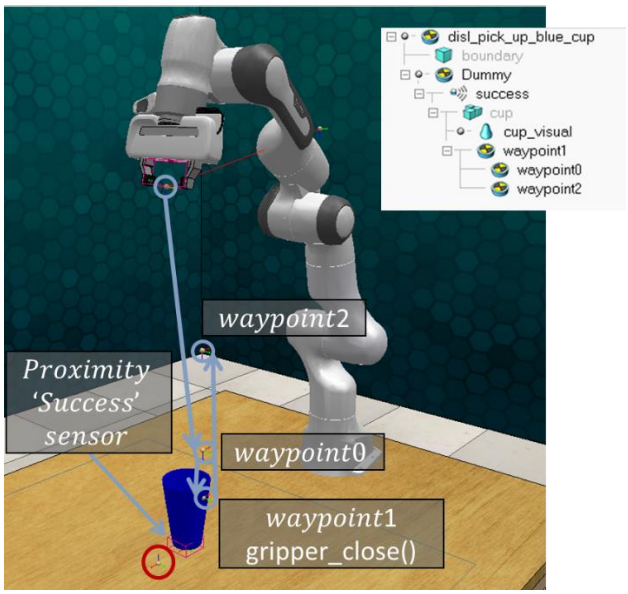


Figure 16 DISL Pick Up Blue Cup Task

This task's CoppeliaSim model is seen in Figure 16 with its full name in the code: DISL Pick Up Blue Cup. The task is defined in a hierarchy seen in the top right of the figure. The "*Dummy*" node, highlighted by the red circle, is the starting point for all tasks and the first element of this task is a proximity sensor

* For a detailed guide on creating a task see: https://github.com/stepjam/RLBench/blob/master/tutorials/simple_task.md

named “*success*.” The *cup* object’s position is defined relative to *success* and the waypoints are defined relative to the cup and each other.

When a demonstration episode is generated, *success*, being a child of *Dummy*, is placed randomly in the workspace. Because the rest of the elements are defined relative to *success* this entire structure gets placed in a new location. The robot begins in the position shown, and the inverse kinematics-solver (IK) attempts to find a path that will align the coordinate frame between the grippers with *waypoint0*. When that position is reached, the IK solver repeats this for a path to *waypoint1* which contains the command to close the gripper. The IK solver commands the robot, now with cup, to *waypoint2*. The demonstration ends when the robot and cup reach the final waypoint.

It is worth noting that RLBench saves simulation time when collecting demonstrations by simplifying the gripping process. Instead of calculating friction forces it rigidly attaches the cup to the end effector and closed the gripper until the finders collide with the cups surface. CoppeliaSim is capable of these contact physics calculations and these could be modeled during reinforcement training.

I constrained the placement of the task such that the cup is only translated across the work area. While picking up the cup from any location around the rim is an equally valid way to lift is, the cup is not rotated between episodes so the waypoints will always be on the robot’s left side. This was done to see how well the network would learn this desired strategy and adhere to it in testing.

This cup pick-up task was created because it represents a real-world task. It requires interacting with the environment and has multiple stages: reaching, grasping, and lifting. The reach target task has one linear motion, but it includes two distractor targets that the network must learn to ignore while reaching for the goal target, which is always colored red.

[9] reports that they achieved an 80% or greater success rate on their task in simulation when their network was trained 200,000 demonstration steps. 800,000 steps were needed to reach that same rate

in the real-world. A typical reach target task has 60 steps, and a typical cup pick-up task has 100 steps. Three datasets were created with this fact in mind, and these are summarized in Table 1.

Table 1 Data Set Summary

Task	Use	Randomized?	Num. of Episodes	Approx. Steps
Cup Pick-Up	Training	No	8000	800,000
Reach Target	Training	No	8000	480,000
Reach Target	Training	Yes	10000	600,000
Cup Pick-Up	Validation	No	20	2,000
Reach Target	Validation	No	20	1,200
Reach Target	Validation	Yes	20	1,200

Since the networks trained for this project are only deployed in simulation, I collected enough episodes to have over double the 200,000-steps that were recommended. The Generate Episodes module of the workflow uses multiprocessing to speed up the collection of these episodes, but it can still be a lengthy process.

On the 4-core machine used for this project each data set took 8 hours to produce and each training set is around 80 GB. These data sizes highlight the issue with how much data is required for a solely behavior cloning approach and why combined methods area appealing.

Training assumes that each episode is contiguously numbered and an interruption in the Generate Episodes code would cause a discontinuity in the data set numbering. The Check Episodes module can fix this by re-numbers the episodes in a data set, starting from 0. The Generate Episodes module can be run again to add new episodes to an existing data set too, finishing the intended data set.

The Check Episodes module also identifies outliers in the data set. While renumbering the episodes, it finds the mean number of steps and standard deviation. Any episodes that deviate by $\pm 2\sigma$ are reported to the user for manual inspection.

The most common failure mode is the IK-solver failing to find a linear path. In the cup pick-up task, 31 of the original 8000 had this issue and were replaced. In these cases, the random placement of the cup put the waypoints too close to the robot's base.

In the third dataset, visual domain randomization is applied to every surface except for the goal and distractor targets.

When an episode is generated, four simulated cameras record the motion in RGB, depth, and a segmented mask. A list records proprioceptive information like joint position, velocity, and torque and the grippers position. When generated, the RGB and depth images are both scaled to $[0,1]$, but when saved the RGB is scaled to the full $[0,255]$ and the depth is encoded in an RGB image. When an episode is loaded from the disc, this process is undone.

4.3 Training Process

After creating a data set, the Imitate module is used to train a network. This program prompts the user to select a training data set and which of the four network types to create: PV, PV-4, RNN PV, or RNN PV-4. It then compiles the desired network and begins the training process. All networks are compiled with the Adam optimizer use MSE loss.

```
 1)  train_order = get_order(num_train, num_epoch)
 2)  total = len(train_order)
 3)  episodes_per_update = 3
 4)
 5)  while episode <= total:
 6)     train_pose = []
 7)     train_view = []
 8)     train_label = []
 9)
10)     for episode in range(episodes_per_update):
11)         pose, view, label = split(format_data(load_data(train_dir,
12)                                                        train_order[step]))
13)         train_pose += pose
14)         train_view += view
15)         train_label += label
16)
17)         episode += 1
18)
19)     network.fit(x=[np.asarray(train_pose),
20)                    np.asarray(train_view)],
21)                 y=np.asarray(train_label),
22)                 batch_size=len(train_pose),
23)                 shuffle=False,
24)                 epochs=1)
```

Algorithm 1 Network Training Pseudocode

Due to the size of the data sets – in the range of 90 GB – it must be loaded into the program's memory in small batches. Because of this, the network must be iteratively fit to the data that is currently available. This process is best explained by viewing Algorithm 1, which describes a condensed version of the training code in the Imitation module.

In line 1, we generate the training order. This is simply a list of integers from 0 to $N - 1$, the number of episodes in that dataset minus one. One randomly shuffled version of this list is returned per epoch requested by the user.

In line 3, I set the number of episodes to load per network update to 3. This was done to prevent the network from overcorrecting itself to fit just a single episode each update. This was done to emulate stochastic gradient descent. The number of episodes to load was selected empirically based on the amount of memory the training process required.

In the training loop, lines 10 to 15 load to memory the next three training episodes. The loaded data is in the form of an RLBench Demonstration object and is directly passed into a function that formats the data. This normalizes the RGB images to the range $[0,1]$ again.

This function also normalizes the joint positions from the range of $[-\pi, \pi]$ to $[0,1]$. This rescaling means that the training labels will be in this range too. In demonstrations the networks predictions must be scaled back to the proper range.

The Demonstration object, now formatted, is passed to a function that splits it into three lists: the pose, the view, and the label.

Table 2 Training Data and Label

Output	**Inputs**	
Label	***Pose***	***View****
$[\theta_1, \dots \theta_7, G]^{(1)}$	$[\theta_1, \dots \theta_7, G]^{(0)}$	$Image^{(0)}$
$[\theta_1, \dots \theta_7, G]^{(2)}$	$[\theta_1, \dots \theta_7, G]^{(1)}$	$Image^{(1)}$
…	…	…
$[\theta_1, \dots \theta_7, G]^{(N)}$	$[\theta_1, \dots \theta_7, G]^{(N-1)}$	$Image^{(N-1)}$
$[\theta_1, \dots \theta_7, G]^{(N)}$	$[\theta_1, \dots \theta_7, G]^{(N)}$	$Image^{(N)}$

Table 2 displays what these lists look like for each episode. The pose list holds an array with the joint angle values and the grippers state. The view list holds the CNN's input either a single RGBD image or a stack of four†. The label is assigned as the next time step's pose. For each of the three episodes that the

* Image matrices are $\in \mathbb{R}^{128\times128\times4}$ for a single image or $\in \mathbb{R}^{128\times128\times16}$ for a stack of four.
† A placeholder of $\mathbf{0} \in \mathbb{R}^{128x128x4}$ is used in the first three steps when a prior image is not available.

network will be trained on, these lists are concatenated to make one master training list of pose, view, and label.

The network fit function, line 19, is part of the TensorFlow API and performs the gradient descent to train the provided data. Because we load the dataset in batches and iteratively call the fit method on the network, a few parameters for this method are changed from the defaults.

On line 22 we set the batch size to be the length of the current training input. The batch size defines how many training data the network should see before updating its parameters. The three episodes that have been loaded from memory *are* the batch, so the network should only update after seeing all of them.

Because the data is time-dependent, order matters. On line 23 the function is told not to reorder the data that is provided. This makes sure that the RNN PV and RNN PV-4 networks see the steps of each episode in the proper sequence.

When the training order is generated on line 1 the data's order is shuffled and the number of epochs is accounted for so on line 34, the number of epochs provided to the method is set to 1. In the context of the fit method, this parameter is how many times it trains over the provided episodes. Setting this to 2, for example, would mean that we train on each set of three episodes twice before moving to the next batch, unintentionally doubling the number of training steps.

In the Imitation code, the fit method returns a dictionary with the training performance. Every so often this performance is saved and added to a list to create a training history. This keeps track of resulting MSE, and the total number of training steps grows.

When complete, the Imitation module saves the network in the behavior cloning data base. The training history is saved as a CSV in the networks folder. The module is also capable of loading a network from this database and training it on further epochs.

4.4 Evaluation and Demonstration

The final two modules, Evaluate and Demonstrate, provide two different means to quantify a network's performance.

Demonstrate, the simpler of the two modules, loads a trained network and launches CoppeliaSim with the correct task environment for that network. As the environment is randomly set up each time the network will not have seen these exact instances before.

The network has 40 steps to complete the task: either placing its gripper on the target or lifting the cup from the proximity sensor, for the reach target and cup pick-up tasks, respectively. The number of successful attempts is recorded.

The success rate is the best method for quantifying a network's ability. But MSE is also a useful metric. The Evaluate module compares a network's prediction to the correct label from episodes in a validation data set.

It is worth noting that the MSE is measured between the correct label network's prediction given the correct pose and image for the step. It is not possible with this code to measure MSE as it accumulates over multiple steps.

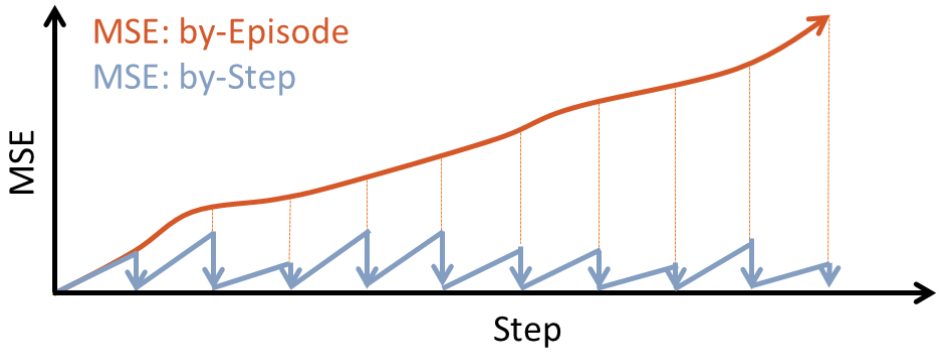


Figure 17 MSE by-Step and by-Episode

This subtle point is illustrated in Figure 17. Evaluation reports the average of the MSE by-step. Collecting the MSE by-episode would start the network at the same initial condition as the validation episode, let it run, and then compare the predictions. This would show how error accumulates over time. While the predicted pose could be feedback into the network, it is not possible to generate the new view from these predicted positions. Instead, after each step, the MSE of the predicted pose is recorded and the correct position and view are to predict the pose for the next step.

Despite this limitation, the relative magnitudes of the validation MSE are still an insightful metric for comparing how networks predict. Even though the average by-step value does not guarantee that the by-Episode error will be small, it stands to reason that a larger by-step would imply poor performance.

5. Results

The workflow presented in Methods was used to train over a dozen neural networks on the reach target and cup pick-up tasks. There were three primary focuses when evaluating these networks. First, what affect does visually domain randomized data have on training? Second, how do the different network structures compare to each other? And third, how does the training time, in epochs, change the network's performance?

The following sections address these questions by comparing how the networks evaluate against the validation training sets. This evaluation focuses on the reach target task. In the final section, I discuss the issues experienced by these networks when demonstrated on their tasks and share thoughts on how this could be resolved.

5.1 Effects of Training Data

As described in Dataset Creation, three datasets were created for training networks. To investigate the effects of visual domain randomization, I train two RNN PV-4 Networks. The first was trained on data set contains 8000 episodes with no domain randomization. And the second used a data set of 10,000 episodes with domain randomization.

These networks were both trained for 5 epochs. Although the data domain randomized data set has more episodes, both data sets have more than double the number total steps that other groups have used for behavior cloning, see 4.2 Dataset Creation. The total steps seen in by each during training places both networks in the range where additional steps do not decrease the training MSE, see Section 5.3 Effects of Training Epochs.

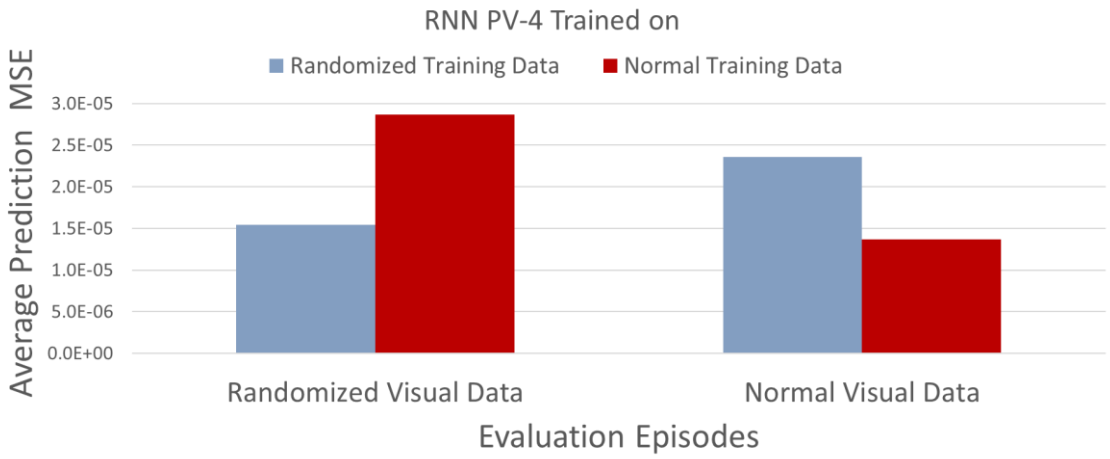


Figure 18 Comparison of Training with Randomized and Normal Visual Domain

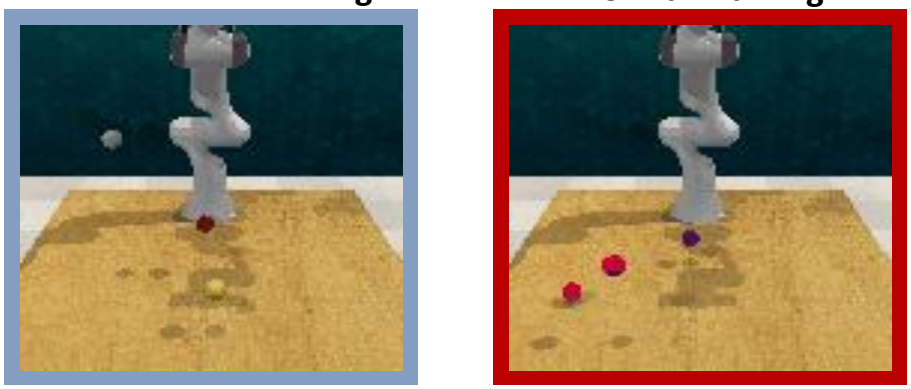
The prediction MSE for these two networks is seen in Figure 18. The two networks were evaluated against two validation data sets. The first was the 20-episode reach target data set *with* domain randomization and second was the 20-episode reach target data set *without* randomization. The average of the 20 episodes average-step MSE is reported.

Unsurprisingly, both networks perform best when evaluated on the same data they were trained on and their MSE for these cases are roughly equal. It is also understandable that the network trained on normal data fares the worst when evaluated on randomized data

As discussed in Sim-to-Real Transfer and Robust Policies, one way to view how the network treats the real world is as a randomized version of the simulated training environment. The network does not care that the world is 'real,' just that the visual input is differs from what it saw in training. One might expect that the network trained on randomized data evaluate better against the normal dataset.

This is where the limitation of MSE as a metric is important. When demonstrating, both networks perform one learned motion repeatedly with little regard to the state of the task, an obvious flaw which is explored in depth in Section 5.4 Demonstrations. Even though these fail the task they do show what motions the network has learned from training.

Randomized Training **Normal Training**



Animation 1 Networks Trained on Randomized and Normal Data

The response of these networks is seen in Animation 1*. Despite its evaluation performance, the network trained on normalized data does not appear to have learned any behaviors. It folds the arm back, away from the targets, and the arm ends up folded behind the base. While the network trained on randomized data is also unsuccessful, it produces a more intelligent motion. It swings its gripper down and toward the middle of the of the work area.

Despite the lack of success, this does indicate that randomized data has helped the network learn a strategy faster. This affirms the consensus of the literature and most of the network created are trained with randomized data.

The domain randomization applied in this project is limited to just the surfaces of objects. But visual randomization could vary more features including the color of the lighting, the position of the lighting, and the position of the camera. More distractor shapes could be placed in the scene.

*Animation figures contain gifs of networks attempting a task. Because these will appear as static images in any format other than Microsoft Word, a verbal description of the gifs is provided after each animation, but it is suggested to view the actual animations here: https://youtu.be/s7PLJI7P-a4

5.2 Effects of Network Structure

The second goal for training networks was to explore how important the image inputs and LSTM layers are for the network. In addition to the two networks trained for Section 5.1 Effects of Training Data, PV, PV-4, and RNN PV networks were trained on the 10,000-episode reach target data set. These networks were also trained over 5 epochs.

After training, the five networks were evaluated against the same two validation data set used in section 5.1 Effects of Training Data.

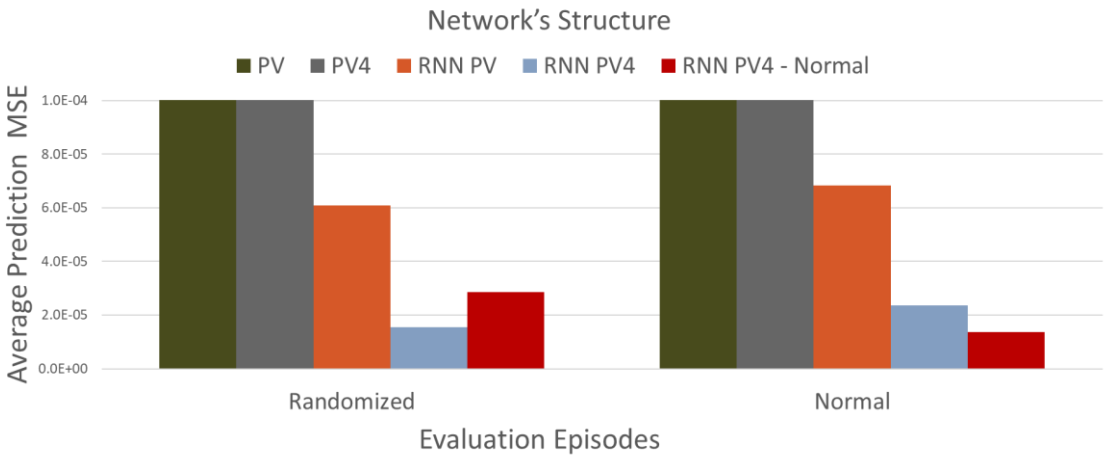
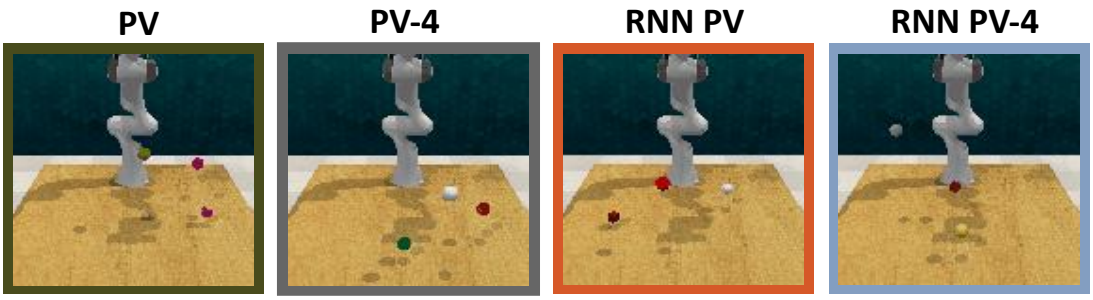


Figure 19 Comparison of Network Structures

The results of this study are seen in Figure 19. It is immediately clear that that the LSTM element is an important aspect for learning a task. While the networks without it could, in theory, achieve the same level of error through given epochs and a larger data set, it appears that it is much more efficient to utilize a LSTM.

Between the RNN PV and RNN PV-4 networks trained with randomized data, it appears that the extra information provided by the stack of four images reduced the MSE by a third when evaluated against both randomized and normal data.

Interestingly, the PV and PV-4 networks had the opposite relationship: the latter performed better. Perhaps without the LSTM layer there is simply too much information for the network to make sense of in the number of steps that it has for training.



Animation 2 Different Networks Trained on the Same Data

Again, the MSE is only a rough indicator of actual performance so Animation 2 shows the actual motion learned by these networks. The RNN PV-4 network is the same as in Animation 1. The PV network has learned sweep to up and around its left and into the middle of the work area while extending its arm. The arm ends up full extended and pointing towards the camera where it oscillates a few times while rotating its wrist. The PV-4 network swings the arm down, towards its left side to place the gripper near its base. And the RNN PV has a similar motion to the PV network.

Again, for any set up of the task environment, each of these networks produced these same motions.

Despite the high MSE, the PV network does learn to reach toward the general region that the targets are in. The successful example of this style of network in [2] was produced by reinforcement learning. It is doubtful that this network could learn a good policy from behavior cloning alone but using it as a starting point for reinforcement could be promising.

Overall, the results indicated that the RNN PV-4 is the best performing of the network structures.

5.3 Effects of Training Epochs

To find out how many training epochs were required to produce a good neural network policy, one RNN PV-4 network was trained on the 10,000-episode reach target task with domain randomization. It was trained for 14 epochs and a copy of the network was saved after 1, 3, 5, and 9 epochs.

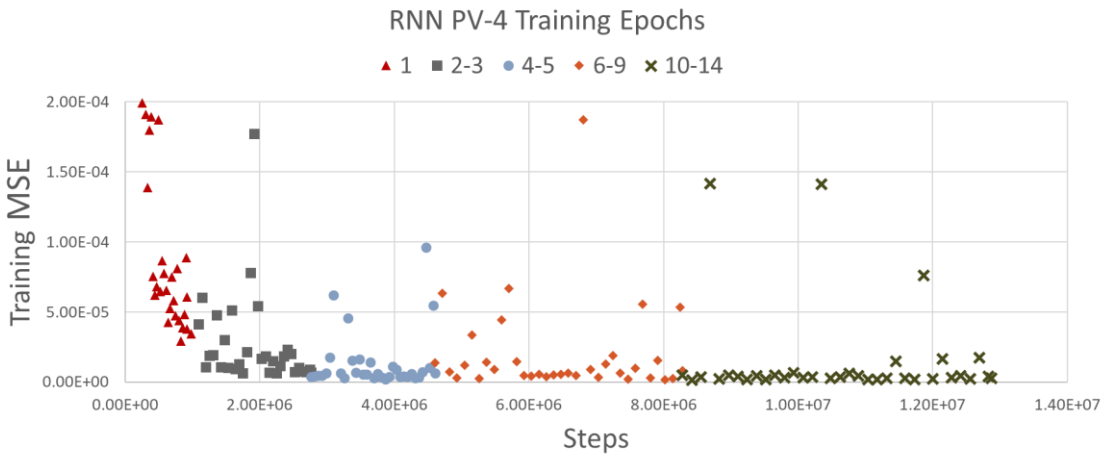
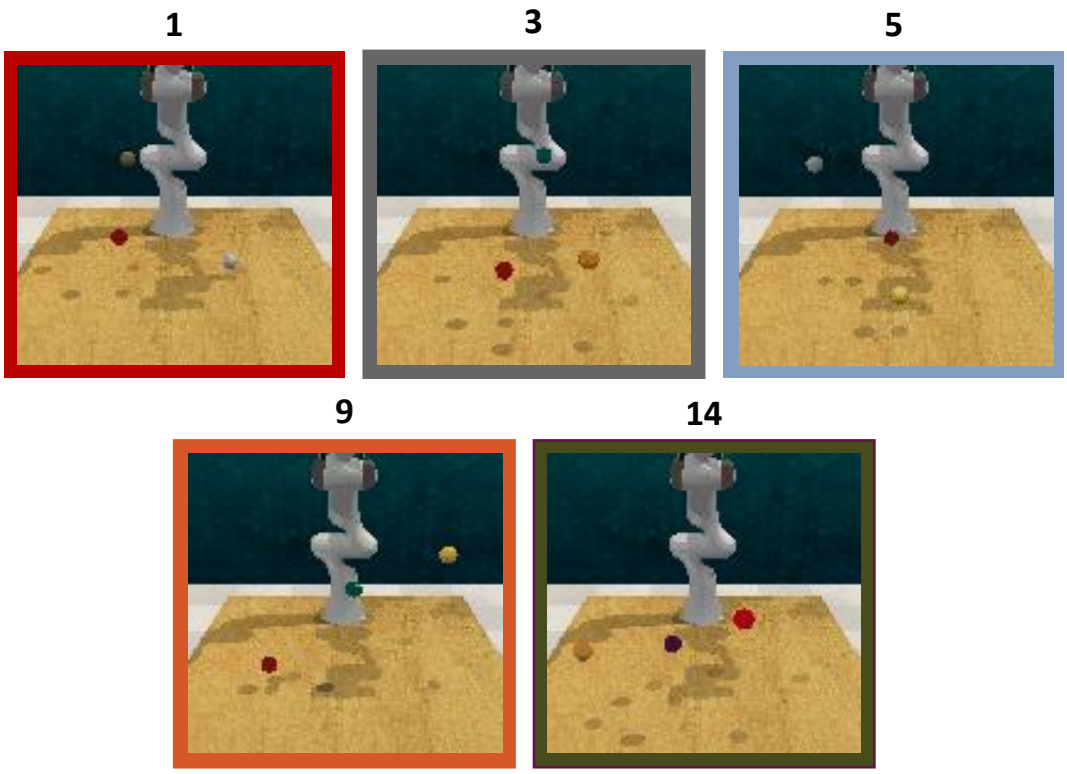


Figure 20 Training MSE over 14 Epochs

The training history shown in Figure 20 Training MSE over 14 Epochs records the MSE at specific training steps. TensorFlow's fit method returns the MSE over the 3 episodes that it just trained the model on and this is recorded at specific intervals while training.

This graph implies that most of the network's learning happens within the first 5 epochs. Additional training does continue to reduce the MSE but at a much slower rate. It is important to know if this additional training time improves the performance in a demonstration or if the network begins to be overfit to the training data.



Animation 3 The Same Network at Different Training Epochs

Through Animation 3 the network's learning process becomes clear. After a single epoch, this network has learned to swing to its left and bring its towards its base, the base rotates the arm to point at the wall to the robot's left. At 3 epochs, the network maintains this leftward swing. But now the gripper's final position is in front of the robot, closer to the work area, and the network has learned not to rotate its base. The motion at 5 epochs stills swings the arm towards the left but now the gripper's final position is higher. After 9 epochs, the network continues this same motion but introduces slight a rotation to its left. The final position of its gripper is further from the work area than the networks trained over 3 and 5 epochs. Finally, at 14 epochs, the network returns to a motion like the 3 and 5 epoch versions, it swings around its left to reach into the middle of the work area. It is notable that this network reaches this position faster than the other networks.

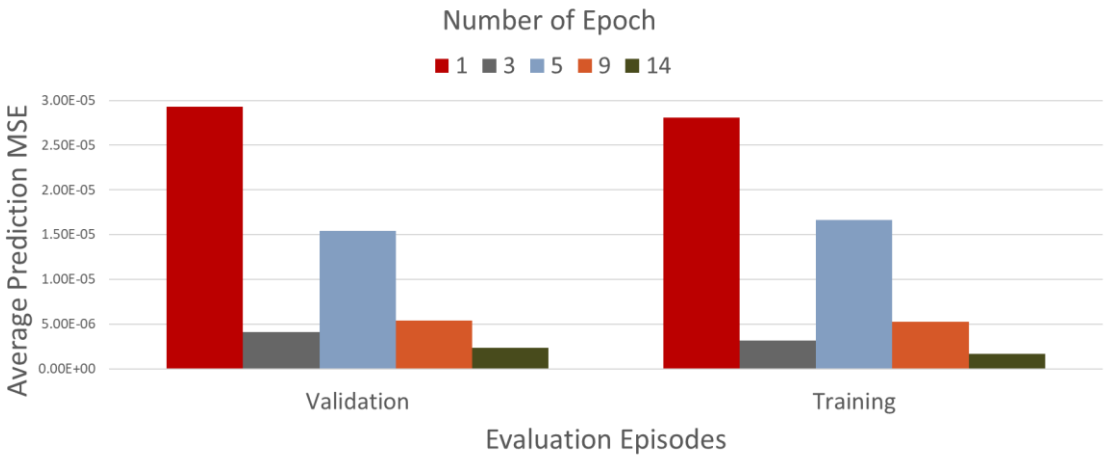


Figure 21 Evaluation MSE After Training for Different Epochs

The MSE for validation data and training data is used to understand how the networks predictions improve and see if additional training has over fit the network. This is shown in Figure 21 where at each network is evaluated first on the same 20-episode validation data set and then on a random 20 episodes from the 10,000-Episode evaluation data set, the average by-step MSE of the 20 episodes is reported.

With the additional training, we see that the network improves at predicting on both the episodes that it has seen during training and those from the validation set that it has never seen. Were a network overfit, it would have a low MSE on the training episodes but perform significantly worse on the unseen validation episodes.

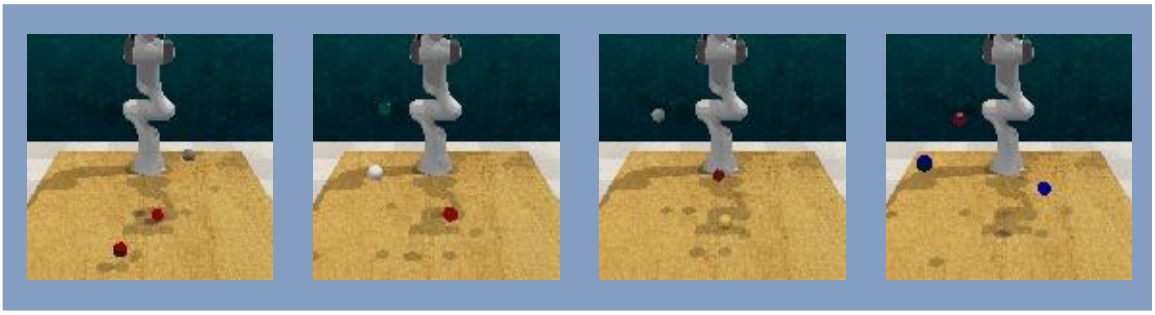
The results indicated that overfitting has not happened, and that additional training will lower the by-step MSE. Interestingly, the network trained on just 3 epochs has the second lowest MSE. The reasons for this were unclear.

Additional training *did not* lead the network to develop new strategies based on the observation of the scene instead of learning some 'average-motion. Despite improved MES values, the training time to reach 14 epochs was around 50 hours. The efficiency of the Imitation module's training process could be

improved, but this is still significant amount of time, so it was deemed impractical to further train this network. Only 3 to 5 epochs seem necessary to learn an approach for the task.

5.4 Demonstrations

As mentioned throughout this chapter, all the networks that were trained presented the same flaw: they repeated the same motion regardless of the position of the goal or distractor targets in the task environment. With my network structures and training process the behavior cloned networks appear to learn some 'average' motion where it reached toward what it believe to be the most likely location of the of the targets.



Animation 4 One Network Attempting Different Episodes of a Task

Animation 4 One Network Attempting Different Episodes of a Task shows the RNN PV-4 network, trained on the 10,000-epriode domain randomized data set for 5 epochs, as it attempts four different instances of the reach target task. The motion is near identical, the exact joint position at each step differs only slightly between each episode. The only visual cue that the motions are not the same is in the oscillation of the grippers in the last few steps--this is most obvious in the leftmost animation.

The networks trained on the 8000-episode cup pick-up task exhibit this same type of behavior.

The natural question is: why do these results differ from other groups that have successfully implemented behavior cloning? There are a few subtle differences in the networks design that may explain the lack of success.

First difference is how the network process the position inputs. The positions and gripper state are fed through 3 dense layers before being concatenated with the output of the final CNN layer. Each dens layer was 64-neuron wide and created a complex set of connection in this branch.

In the behavior cloning example from literature, [9], this branch was directly concatenated with the CNN's output without going through any DNN layers. This position network structure I use was inspired by another group [2] that had trained their network via reinforcement learning.

From the animations, it seems that the network does not value the CNN's contributions very much; each demonstration starts the robot in the same position and then performs the same motion. Across episodes, no matter what the visual input is, the initial position input is the same and the predictions are the same.

In training, the complete network might over-rely on the complex position network. Instead of learning to achieve the task, it may have ignored the visual input and just learned the pattern for how the arm moves. Because the predictions for how the arm might move each step are good, the network scores well in by-step MSE but fails in actual demonstrations of an episode.

This structure may perform well when trained by reinforcement learning, however, because the repetition of action and reward could guide the complete network to learn that the visual inputs are important for completing the task. But, through behavior cloning alone, simply learning to repeat the expert demonstration's motions between steps might not imply that the *actual task* was learned.

Another difference between my networks and the behavior cloned example in literature [9] is what the networks predicted. In addition to predicting the joint velocities, their network also included several auxiliary outputs.

These auxiliary outputs predict the position of the robot's gripper and the position of the cube that it is reaching. These predictions are not used when the network is running, instead they encode other task-relevant information into the training by adding to the MSE. The intuition is that by minimizing the MSE with these auxiliary outputs, the network learns that these features are important to the task.

To simplify the network structure in this project, these kinds of auxiliary outputs were omitted. However, this might have been in error and these could be the key to learning completely by behavior cloning.

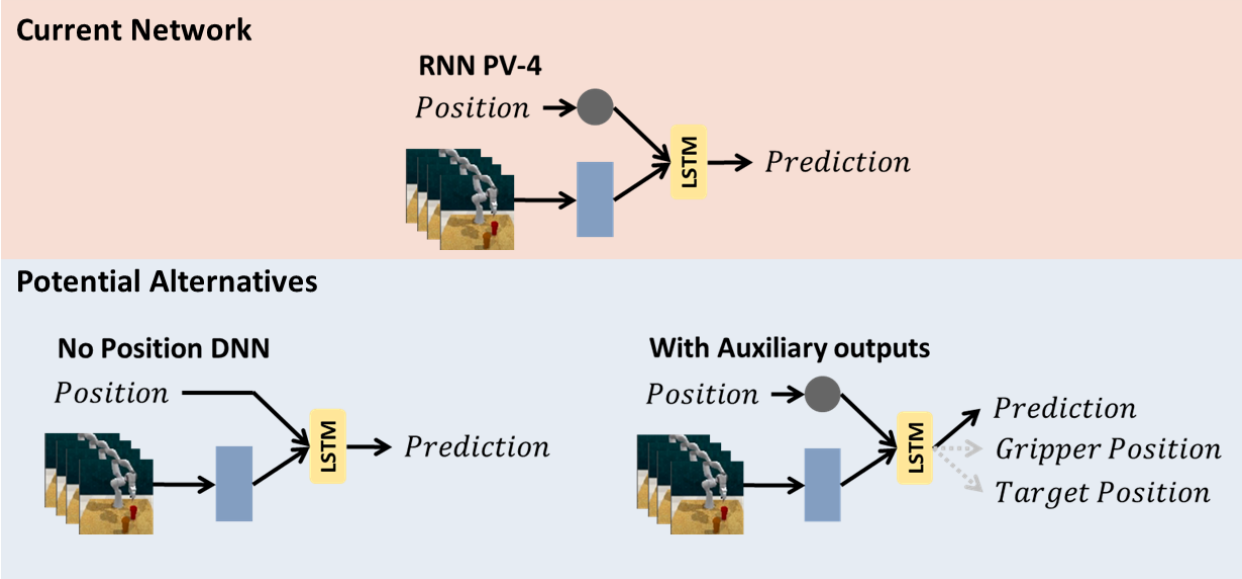


Figure 22 Potential Network Alteration

The networks in Figure 22 compare the current structure with the two alterations mentioned above. Comparing how the network performs with one or both changes would provide important insight on how the network learns.

Networks were also trained on the cup pick-up task. However, these experienced similar issues to the ones trained on the reach target task.

6. Conclusion and Future Work

In this project I have presented a workflow for training neural network visuomotor policies by behavior cloning. This workflow in just the initial work on a more complete process that includes modules for reinforcement learning and aids in deploying these models to real-world robots.

For the reach target task, I compared how variations on one neural network structure perform when trained through behavior cloning. The results showed that the RNN PV-4 structure performed best and that between 3 to 5 epochs of training on the 10,000-episode data set provided enough training for the network to learn to reach toward the work area.

This comparison of networks indicated that the inclusion of an LSTM layer is important in reducing the by-step MSE of the network. Additionally, using a stack of the current and previous RGBD images as the input to the network appears to further reduce the MSE.

The performance of the networks on live demonstrations, however, implied that the networks may over rely on the position branch of the network and learned something other than the task. Section 5.4 Demonstrations provides some ideas for altering the RNN PV-4 structure to properly learn the task.

These changes should be investigated in the next steps of this project. The first suggestion was reducing the complexity of the DNN that the position input passes through or removing the DNN completely to pass these values directly to LSMT layer. And the second suggestion was adding auxiliary outputs that describe more of the robot's or the environment's state.

Completing the workflow's Reinforce module with a program that can train a new or existing behavior cloned model is also an important step. RLBench already provides the ability to generate new task environments for the training and a reward function can be defined when creating a task – or added to an existing one.

Creating this reinforcement module will require selecting the algorithm to use. Proximal Policy Optimization (PPO), Deep Deterministic Policy Gradient (DDPG), or Asynchronous Advantage Actor-Critic Algorithm (A3C) are popular algorithms which can be used with continuous action and state space.

Once this is reinforcement module is completed, there should be an evaluation of the training processes to identify how behavior cloning affects the reinforcement learning. The goal of this would be to identify how much behavior cloning pre-training could improve the training speed or task completion rate for reinforcement learning.

Finally, when the simulation training process is well understood, these networks will be transferred to the real-world. In this project the Franka Panda is used because it is the default configuration for RLBench. But this can be changed with minor alterations to the code and the tasks. The target platform is a Universal Robots UR5 with a custom-designed gripper and Intel Real Sense depth camera.

A model of this gripper in CoppeliaSim will need to be made and added to PyRep[*] and then RLBench[†]. The UR5 is supported in both already. And a custom task will need to be created to generate training demonstrations and to provide a task environment for reinforcement learning. Once trained, a Robotic Operating System (ROS) packaged could be created to set up a connection between the robot, camera, and the computer running the network.

[*] Grippers and robots are added in the same manner, see: https://github.com/stepjam/PyRep/blob/master/tutorials/adding_robots.md

[†] For more details see https://github.com/stepjam/RLBench/issues/111

References

[1] B. Heater, “Amazon built an electronic vest to improve worker/robot interactions,” *TechCrunch*, Jan. 18, 2019.

[2] L. Hermann, M. Argus, A. Eitel, A. Amiranashvili, W. Burgard, and T. Brox, “Adaptive Curriculum Generation from Demonstrations for Sim-to-Real Visuomotor Control,” *ArXiv191007972 Cs*, Jul. 2020, Accessed: Mar. 14, 2021. [Online]. Available: http://arxiv.org/abs/1910.07972.

[3] X. B. Peng, M. Andrychowicz, W. Zaremba, and P. Abbeel, “Sim-to-Real Transfer of Robotic Control with Dynamics Randomization,” in *2018 IEEE International Conference on Robotics and Automation (ICRA)*, Brisbane, QLD, May 2018, pp. 3803–3810, doi: 10.1109/ICRA.2018.8460528.

[4] S. Gu, E. Holly, T. Lillicrap, and S. Levine, “Deep Reinforcement Learning for Robotic Manipulation with Asynchronous Off-Policy Updates,” *ArXiv161000633 Cs*, Nov. 2016, Accessed: Mar. 14, 2021. [Online]. Available: http://arxiv.org/abs/1610.00633.

[5] K. Arndt, M. Hazara, A. Ghadirzadeh, and V. Kyrki, “Meta Reinforcement Learning for Sim-to-real Domain Adaptation,” *ArXiv190912906 Cs*, Sep. 2019, Accessed: Mar. 14, 2021. [Online]. Available: http://arxiv.org/abs/1909.12906.

[6] M. Kaspar, J. D. M. Osorio, and J. Bock, “Sim2Real Transfer for Reinforcement Learning without Dynamics Randomization,” *ArXiv200211635 Cs*, Feb. 2020, Accessed: Mar. 14, 2021. [Online]. Available: http://arxiv.org/abs/2002.11635.

[7] Y. Zhu *et al.*, “Reinforcement and Imitation Learning for Diverse Visuomotor Skills,” *ArXiv180209564 Cs*, May 2018, Accessed: Mar. 14, 2021. [Online]. Available: http://arxiv.org/abs/1802.09564.

[8] A. Rajeswaran *et al.*, “Learning Complex Dexterous Manipulation with Deep Reinforcement Learning and Demonstrations,” *ArXiv170910087 Cs*, Jun. 2018, Accessed: Mar. 14, 2021. [Online]. Available: http://arxiv.org/abs/1709.10087.

[9] S. James, A. J. Davison, and E. Johns, “Transferring End-to-End Visuomotor Control from Simulation to Real World for a Multi-Stage Task,” *ArXiv170702267 Cs*, Oct. 2017, Accessed: Mar. 14, 2021. [Online]. Available: http://arxiv.org/abs/1707.02267.

[10] E. Coumans, “Bullet physics simulation,” in *ACM SIGGRAPH 2015 Courses*, Los Angeles California, Jul. 2015, p. 1, doi: 10.1145/2776880.2792704.

[11] E. Todorov, T. Erez, and Y. Tassa, “MuJoCo: A physics engine for model-based control,” in *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, Vilamoura-Algarve, Portugal, Oct. 2012, pp. 5026–5033, doi: 10.1109/IROS.2012.6386109.

[12] S. James, Z. Ma, D. Rovick Arrojo, and A. Davison, “RLBench: The Robot Learning Benchmark & Learning Environment,” *ArXiv*, Sep. 2019, [Online]. Available: https://arxiv.org/abs/1909.12271.

[13] E. Rohmer, S. P. N. Singh, and M. Freese, “V-REP: A versatile and scalable robot simulation framework,” in *2013 IEEE/RSJ International Conference on Intelligent Robots and Systems*, Tokyo, Nov. 2013, pp. 1321–1326, doi: 10.1109/IROS.2013.6696520.

[14] S. James, M. Freese, and A. J. Davison, “PyRep: Bringing V-REP to Deep Robot Learning,” *ArXiv190611176 Cs*, Jun. 2019, Accessed: Mar. 14, 2021. [Online]. Available: http://arxiv.org/abs/1906.11176.

[15] M. Abadi *et al.*, “TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems,” *ArXiv160304467 Cs*, Mar. 2016, Accessed: Mar. 17, 2021. [Online]. Available: http://arxiv.org/abs/1603.04467.

[16] S. Ruder, “An overview of gradient descent optimization algorithms,” *rudder.io*. https://ruder.io/optimizing-gradient-descent/.

[17] N. Cui, “Applying Gradient Descent in Convolutional Neural Networks,” *J. Phys. Conf. Ser.*, vol. 1004, p. 012027, Apr. 2018, doi: 10.1088/1742-6596/1004/1/012027.

[18] C. Olah, “Understanding LSTM Networks,” *colah’s blog*. https://colah.github.io/posts/2015-08-Understanding-LSTMs/.
[19] K. Fang *et al.*, “Learning Task-Oriented Grasping for Tool Manipulation from Simulated Self-Supervision,” *ArXiv180609266 Cs Stat*, Jun. 2018, Accessed: Mar. 14, 2021. [Online]. Available: http://arxiv.org/abs/1806.09266.
[20] J. Tremblay, T. To, B. Sundaralingam, Y. Xiang, D. Fox, and S. Birchfield, “Deep Object Pose Estimation for Semantic Robotic Grasping of Household Objects,” *ArXiv180910790 Cs*, Sep. 2018, Accessed: Mar. 24, 2021. [Online]. Available: http://arxiv.org/abs/1809.10790.
[21] K. Hausman, Y. Chebotar, S. Schaal, G. Sukhatme, and J. Lim, “Multi-Modal Imitation Learning from Unstructured Demonstrations using Generative Adversarial Nets,” *ArXiv170510479 Cs*, Nov. 2017, Accessed: Mar. 16, 2021. [Online]. Available: http://arxiv.org/abs/1705.10479.
[22] A. Hussein, M. M. Gaber, E. Elyan, and C. Jayne, “Imitation Learning: A Survey of Learning Methods,” *ACM Comput. Surv.*, vol. 50, no. 2, pp. 1–35, Jun. 2017, doi: 10.1145/3054912.
[23] W. Zhao, J. Pena Queralta, and T. Westerlund, “Sim-to-Real Transfer in Deep reinforcement Learning for Robotics: a Survey,” *ArXiv*, Sep. 2020, [Online]. Available: https://arxiv.org/abs/2009.13303.

List of Figures